

# SeedTree: A Dynamically Optimal and Local Self-Adjusting Tree

Arash Pourdamghani<sup>1</sup>, Chen Avin<sup>2</sup>, Robert Sama<sup>3</sup>, Stefan Schmid<sup>1,4</sup>

<sup>1</sup>TU Berlin, Germany <sup>2</sup>School of Electrical and Computer Engineering, Ben Gurion University of the Negev, Israel

<sup>3</sup>Faculty of Computer Science, University of Vienna, Austria <sup>4</sup>Fraunhofer SIT, Germany

**Abstract**—We consider the fundamental problem of designing a self-adjusting tree, which efficiently and locally adapts itself towards the demand it serves (namely accesses to the items stored by the tree nodes), striking a balance between the benefits of such adjustments (enabling faster access) and their costs (reconfigurations). This problem finds applications, among others, in the context of emerging demand-aware and reconfigurable datacenter networks and features connections to self-adjusting data structures. Our main contribution is *SeedTree*, a dynamically optimal self-adjusting tree which supports local (i.e., greedy) routing, which is particularly attractive under highly dynamic demands. *SeedTree* relies on an innovative approach which defines a set of unique paths based on randomized item addresses, and uses a small constant number of items per node. We complement our analytical results by showing the benefits of *SeedTree* empirically, evaluating it on various synthetic and real-world communication traces.

**Index Terms**—Reconfigurable datacenters, Online algorithms, Self-adjusting data structure

## I. INTRODUCTION

This paper considers the fundamental problem of designing self-adjusting trees: trees which adapt themselves towards the demand they serve. Such self-adjusting trees need to strike an efficient tradeoff between the benefits of such adjustments (better performance in the future) and their costs (reconfiguration overheads now). The problem is motivated by the fact that workloads in practice often feature much temporal and spatial structure, which may be exploited by self-adjusting optimizations [1], [2]. Furthermore, such adjustments are increasingly available, as researchers and practitioners are currently making great efforts to render networked and distributed systems more flexible, supporting dynamic reconfigurations, e.g., by leveraging programmability (via software-defined networks) [3], [4], network virtualization [5], or reconfigurable optical communication technologies [6].

In particular, we study the following abstract model (applications will follow): we consider a binary tree which serves access requests, issued at the root of the tree, to the items stored by the nodes. Each node (e.g., server) stores up to  $c$  items (e.g., virtual machines), where  $c$  is a parameter indicating the *capacity* of a node. We consider an online perspective where items are requested over time. An online algorithm aims to optimize the tree in order to minimize the cost of future access requests (defined as the path length

between root and accessed item), while minimizing the number of items moving up or down in the tree: the *reconfigurations*. We call each movement a reconfiguration, and keep track of its cost. In particular, the online algorithm which does not know the future access requests, aims to be competitive with an optimal offline algorithm that knows the entire request sequence ahead of time. In other words, we are interested in an online algorithm with minimum *competitive ratio* [7] over any (even worst-case) request sequence.

Self-adjusting trees are not only one of the most fundamental topological structures of their own merit, they also have interesting applications. For example, such trees are a crucial building block for more general self-adjusting networks: Avin et al. [8] recently showed that multiple trees optimized individually for a single root, can be combined to build general communication networks which provide low degree and low distortion. The design of a competitive self-adjusting tree as studied in this paper, is hence a stepping stone.

Self-adjusting trees also feature interesting connections to self-adjusting data structures (see §VI for a detailed discussion), for some of which designing and proving constant-competitive online algorithms is still an open question [9]. Interestingly, a recent result shows that constant-competitive online algorithms exist for self-adjusting balanced binary trees if one maintains a *global* map of the items in the tree; it was proposed to store such a map centrally, at a logical root [10]. In this paper, we are interested in the question whether this limitation can be overcome, and whether a competitive *decentralized* solution exist.

Our main contribution is a dynamically optimal self-adjusting tree, *SeedTree*<sup>\*</sup>, which achieves a constant competitive ratio by keeping recently accessed items closer to the root, ensuring a working set theorem [9]. Our result also implies weaker notions such as key independent optimality [11] (details will follow). *SeedTree* further supports *local* (that is, greedy and hence decentralized) routing, which is particularly attractive in dynamic networks, by relying on an innovative and simple routing approach that enables nodes to take local forwarding decisions: *SeedTree* hashes items to *i.i.d.* random addresses and defines a set of greedy paths based on these addresses. A main insight from our work is that a constant competitive ratio with locality property can be achieved if

This project has received funding from the European Research Council (ERC) under grant agreement No. 864228 (AdjustNet), 2020-2025.

<sup>\*</sup>The name is due to the additional capacity in nodes of the tree, which resembles seeds in fruits of a tree.

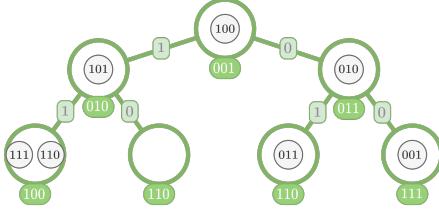


Fig. 1: A depiction of *SeedTree* with capacity 2. Large circles represent nodes (nodes) of the system, and small circles represent items. The number inside each small circle is the hash of the corresponding item.

nodes feature small constant *capacities*, that is, by allowing nodes to store a small constant number of items. Storing more than a single item on a node is often practical, e.g., on a server or a peer [12], and it is common in hashing data structures with collision [13], [14]. We also evaluate *SeedTree* empirically, both on synthetic traces with ranging temporal locality and also data derived from Facebook datacenter networks [1], showing how tuning parameters of the *SeedTree* can lower the total (and access) cost for various scenarios.

The remainder of the paper is organized as follows. §II introduces our model and preliminaries. We present and analyze our online algorithm in §III, and transform it to the matching model of datacenter networks in §IV. After discussing our empirical evaluation results in §V, we review related works in §VI and conclude our contributions in §VII.

## II. MODEL AND PRELIMINARIES

This section presents our model and introduces preliminaries used in the design of *SeedTree*.

**Items and nodes.** We assume a set of *items*  $V = (v_1, \dots, v_n)$ , and a set of *nodes*  $S = (s_1, \dots)$ <sup>†</sup> arranged as a binary tree  $T$ . We call the node  $s_1$  the root, which is at *depth* 0 in the tree  $T$ , and a node  $s_j$  is at depth  $\lfloor \log j \rfloor$ .

Each node can store  $c$  items, where  $c$  is a parameter indicating the *capacity* of a node. In our model, we assume that  $c$  is a constant. The assignment of items to nodes can change over time. We say a node is *full* if it contains  $c$  items, and *empty* if it contains no item (See an example in Figure 1).

We define the level of item  $v$  at time  $t$ ,  $\text{level}_t(v)$ , as the depth of the node containing  $v$ . For example, if item  $v$  is at node  $s_j$  at time  $t$ , we have  $\text{level}_t(v) = \lfloor \log j \rfloor$ .

**Request Sequence and Working Set.** Items are requested over time in an online manner, modeled as a *request sequence*  $\sigma = (\sigma_1, \dots, \sigma_m)$ , where  $\sigma_t = v \in V$  means item  $v$  is requested at time  $t$ . We are sometimes interested in the recency of item requests, particularly the size of the working set. Formally, we define  $ws_t(\sigma, v)$  as the *working set* of item  $v$  in at time  $t$  in the request sequence  $\sigma$ . The working set  $ws_t(\sigma, v)$  is a set of unique items requested since the last request to the item  $v$  before time  $t$ . We define a *rank* of item  $v$  at time  $t$ ,  $\text{rank}_t(v)$ , as the size of working set of the item  $v$  at time  $t$ .

<sup>†</sup>We assume the set of nodes to be arbitrarily large, as the exact number of nodes will be determined based on their used capacity.

**Costs and Competitive Ratio.** We partition costs incurred by an algorithm,  $ALG$ , into two parts, the cost of finding an item: the *access cost*, and the cost of reconfigurations: the *reconfiguration cost*. The search for any item starts at the root node and ends at the node containing the item. Based on our assumption of constant capacity, we assume the cost of search inside a node to be negligible. Furthermore, assuming the local routing property, we find an item by traversing a single path in our tree; hence the access cost for an access request  $\sigma_i$ ,  $C_{ALG}^A(\sigma_i)$ , equals the level at which the item is stored.

In our model, a reconfiguration consists of moving an item one level up or one level down in the tree, plus potentially additional lookups inside a node. We denote the total reconfiguration cost after an access request  $\sigma_i$  by  $C_{ALG}^R(\sigma_i)$ . Hence, the total cost of each access request is  $C_{ALG}^A(\sigma_i) + C_{ALG}^R(\sigma_i)$ , and the total cost of the algorithm on the whole request sequence is:  $C_{ALG}(\sigma) = \sum_{i=1}^m C_{ALG}^A(\sigma_i) + C_{ALG}^R(\sigma_i)$ . The objective of *SeedTree* is to operate at the lowest possible cost, or more specifically, as close as possible to the cost of an optimal offline algorithm,  $OPT$ .

**Definition 1** (Competitive ratio). *Given an online algorithm  $ALG$  and an optimal offline algorithm  $OPT$ , the (strict) competitive ratio is defined as:  $\rho_{ALG} = \max_{\sigma} \frac{C_{ALG}(\sigma)}{C_{OPT}(\sigma)}$*

Furthermore, we say an algorithm has (strict) *access competitive ratio* considering only the access cost of the online algorithm  $ALG$  (not including the reconfiguration cost).

In this paper, we prove that *SeedTree* is *dynamically optimal*. It means that the cost of our algorithm matches the cost of the optimal offline algorithm asymptotically.

**Definition 2** (Dynamic optimality). *Algorithm  $ALG$  is dynamically optimal if it has constant competitive ratio, i.e.,  $\rho_{ALG} = O(1)$ .*

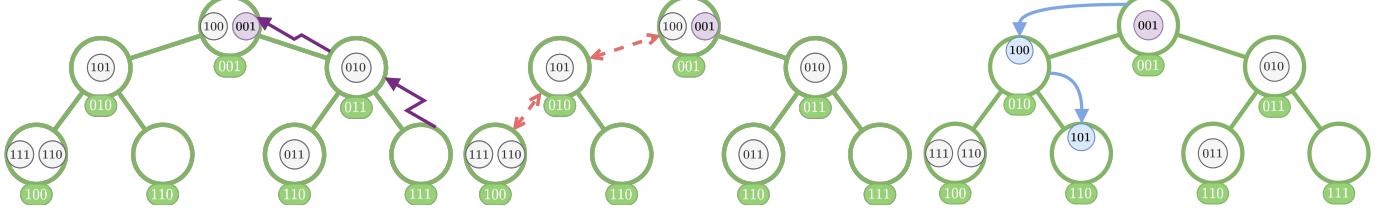
**MRU trees.** We define a specific class of self-adjusting trees, *MRU trees*. An algorithm maintains a *MRU tree* if it keeps items at a similar level to their ranks.

**Definition 3** (*MRU tree*). *An algorithm has the  $MRU(0)$  property if for any item  $v$  inside its tree and at any given time  $t$ , the equality  $\text{level}_t(v) = \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rceil$  holds.*

*Similarly, we say an algorithm maintains an  $MRU(\beta)$  if it ensures the relaxed bound of  $\text{level}_t(v) \leq \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rceil + \beta$  for any item  $v$  in the tree.*

## III. ONLINE SEEDTREE

This section presents *SeedTree*, an online algorithm that is dynamically optimal in expectation. This algorithm build upon uniformly random generated addresses, and allows for local routing, while ensuring dynamic optimality. Details of the algorithm are as follows: Algorithm 1 always starts from the root node. Upon receiving an access request to an item  $v$  it performs a local routing (described in Procedure LocalRouting) based on the uniformly random binary address generated for the node  $v$ , which uniquely determines the path of  $v$  in the tree. We call the  $i$ -th bit of the address of  $v$  by  $H(v, i)$ . Let us assume that the local routing for node  $v$  ends in level  $\ell$ .



(a) Item 001 moves up, node-by-node, until it reaches the root.  
(b) The first try of push-down failed, because node 100 is full.  
(c) After finding non-full node, items are pushed down node-by-node.

Fig. 2: An example of steps taken in Algorithm 1, starting from the state of *SeedTree* in Figure 1, which has a capacity equal to 2. In this example, the request is an access request to the item with the hash value 001 (the purple circle). Subfigure 2a shows the move-to-the-root phase, and Subfigures 2b and 2c depict the push-down phase.

---

**Procedure** LocalRouting( $s, v$ )

---

```

1 if  $H(v, \text{level}(s))$  equals 0 then
2   Return the left child of  $s$ .
3 else
4   Return the right child of  $s$ .

```

---

Then *SeedTree* performs the following two-phase reconfiguration. These two phases are designed to ensure the level of items remains in the same range as their rank (details will follow), and the number of items remains the same at each level.

- 1) *Move-to-the-root*: This phase moves the accessed item to the node at the lowest level possible, the root of the tree. The movement of the item is step-by-step, and it keeps all the other items in their previous node (we keep the item in a temporary buffer if a node on the path was full). This phase is depicted in Figure 2a by zig-zagged purple arrows.
- 2) *Push-down*: In this phase, our algorithm starts from the root node, selects an item in the node (including the item that has just moved to this node) uniformly at random, and moves this item one level down to the new node selected in the LocalRouting procedure. The same procedure is continued for the new node until reaching level  $\ell$ , the level of the accessed item. If the node at level  $\ell$  was non-full, the re-establishment of balance was successful. Otherwise, if this attempt is failed, the algorithm reverses the previous push downs back to the root, and starts again, until an attempt is successful. As an example, the failed attempt of this phase is depicted by dashed red edges in Figure 2b and the last successful one by curved blue arrows in Figure 2c.

Algorithm 1 always terminates, as there is always the chance that the item which has been moved to root is selected among all candidates, and we know that the node which that item is taken from is not full. We now state the main theorem of the paper that proves the dynamic optimality of *SeedTree*.

**Theorem 1.** *SeedTree* is dynamically optimal for any given capacity  $c \geq 1$ .

---

**Algorithm 1:** Online *SeedTree*


---

```

Input: Accessed item  $v$ .
1 Set  $s$  as the root.
2 while  $s$  does not contain  $v$  do
3    $s = \text{LocalRouting}(s, v)$ .
4 Call the current level of  $v$  as  $\ell$ .
5 Set  $s$  as the root, and move item  $v$  to  $s$ .
6 while balance is not fixed do
7   Call the current node  $s$ .
8   while level of  $s$  is less than  $\ell$  do
9     Take an item in node  $s$ , uniformly at random, call it  $v$ .
10     $s = \text{LocalRouting}(s, v)$ .
11    Add item  $v$  to the node  $s$ .
12   if the last chosen node is full then
13     Reverse the push-down back to the root.

```

---

The proof of Theorem 1 is at the end of the section. The first step towards the proof is showing that the number of items in each level remains the same. It is true because after removing an item at a certain level, the algorithm adds an item to the same level as a result of the push-down phase.

**Observation 1.** *SeedTree* keeps the number of items the same at each level.

The rest of the analysis is based on the assumption that the algorithm was initialized with a fixed *fractional occupancy*  $0 < f < 1$  of the capacity of each level, i.e., in level  $i$ , the initial tree has exactly  $\lfloor c \cdot f \cdot 2^i \rfloor$  items. At the end of this section, we will see that  $f = \frac{1}{2}$  works best for our analysis. However, we emphasize that having  $0 < f < 1$  suffices for *SeedTree* to run properly.

The second observation is a result of Observation 1. As the number of items remains the same in each level (based on Observation 1) at most a fraction  $f$  of all nodes are full. In the lowest level, the number of full nodes might be even lower; hence the probability of a uniformly random node being full is at most  $f$  when we go to the next request.

**Observation 2.** Algorithm 1 ensures that the probability of any uniformly random chosen node in *SeedTree* to be full, after serving each access request, is at most  $f$ .

According to Algorithm 1, items are selected uniformly at random inside a node. In the following lemma, we show that a node in a certain level is also selected uniformly at random, which enables the rest of the proof.

**Lemma 1.** *Nodes selected on the final path of the push-down phase with a level lower than  $\ell$  are selected uniformly at random.*

*Proof.* Let us denote the probability of  $\ell'$ -th node on the path (the node at level  $\ell'$ , denoted by  $s_{\ell'}$ ) being the selected node is  $\frac{1}{2^{\ell'}}$ . Our proof goes by induction. For the basis,  $\ell' = 0$ , it is true since we only have one node, the root. Now assume that in the final path of push down, we want to see the probability of reaching the current node,  $s_{\ell'}$ . Based on the induction assumption, we know that the parent of  $s_{\ell'}$ , the node  $s_{\ell'-1}$ , has been selected uniformly at random, with probability  $\frac{1}{2^{\ell'-1}}$ . Based on Line 9 of Algorithm 1, an item is selected from those inside  $s_{\ell'-1}$  uniformly at random, plus having the independence guarantee of our hash function that generated address of the selected item, we can conclude the decision to go to left or right from  $s_{\ell'-1}$  was also uniformly at random, hence the probability of reach  $s_{\ell'}$  is  $\frac{1}{2^{\ell'-1}} \cdot \frac{1}{2} = \frac{1}{2^{\ell'}}$ . Note that the above-mentioned choices are independent of whether or not the descents  $s_{\ell'-1}$  are full or not. Hence the choice is independent of (possible) previous failed attempts of the push-down phase (which might happen due to having a full node at level  $\ell$ ), i.e., the previous attempts do not affect the probability of choosing the node  $s_{\ell'}$ .  $\square$

An essential element of the proof of Theorem 1 is that the rank and level of items are related to each other. Lemma 2 describes one of the aspects of this relation.

**Lemma 2.** *During the execution of the SeedTree, for items  $v$  and  $u$  at time  $t$ , if  $\text{rank}_t(v) > \text{rank}_t(u)$  then  $E[\text{level}_t(v)] > E[\text{level}_t(u)]$ .*

*Proof.* Having  $\text{rank}_t(v) > \text{rank}_t(u)$ , we know that  $u$  was accessed more recently than  $v$ . Let us consider time  $t'$ , the last time  $u$  was accessed. Since the rank of  $v$  is strictly larger than the rank of  $u$ , and as  $u$  was moved to the root at time  $t'$ , we know that  $\text{level}_{t'}(v) > \text{level}_{t'}(u)$ .

Items  $u$  and  $v$  might reach the same level after time  $t'$ , but it is not a must. We consider the level that they first met as a random variable,  $L_{uv}$ . We denote  $L_{uv} = -1$  if  $u$  and  $v$  never appear on the same level after time  $t'$ . Let us quantify the difference in the expected level of  $u$  and  $v$ , using the law of total expectation:

$$\begin{aligned} & E[\text{level}_t(v)] - E[\text{level}_t(u)] \\ &= \sum_{k=-1}^{\lfloor \log \lceil \frac{n}{c} \rceil \rfloor} Pr(L_{uv} = k) \cdot (E[\text{level}_t(v)|L_{uv} = k] \\ &\quad - E[\text{level}_t(u)|L_{uv} = k]) \end{aligned}$$

For the case  $L_{uv} = -1$ , we know that  $u$  and  $v$  never reached the same level, and the following is always true:

$$E[\text{level}_t(v)|L_{v,u} = -1] > E[\text{level}_t(u)|L_{v,u} = -1]$$

For  $k \geq 0$ , let us consider the time  $t''$  when  $u$  and  $v$  meet at the same level, i.e  $\text{level}_{t''}(u) = \text{level}_{t''}(v)$ . After items  $u$  and  $v$  meet for the first time, their expected progress is the same. More precisely, consider the current subtree of the node containing  $v$  at time  $t''$ , and call it  $T'$ . Since the item addresses are chosen uniformly at random, the expected number of times that  $T'$  is a subtree of a node containing  $v$ , equals the number of times that  $T'$  might be a subtree of node containing  $u$  in the same level. Hence the expected increase in the level for both items  $u$  and  $v$  stays the same from time  $t''$  onward.  $\square$

Next, we explain why the number of items accessed at a higher level is limited in expectation for any given item.

**Lemma 3.** *For a given item  $v$  at time  $t$ , there are at most  $2 \cdot \text{rank}_t^{ws}(v)$  items accessed at a higher level since the last time  $v$  was accessed, in expectation.*

*Proof.* Given Lemma 2, the proof is along the lines of the proof of Lemma 4 from [10]. We removed the details of the proof due to space constraints.  $\square$

Now we prove the items in the tree maintained by the online *SeedTree* are not placed much farther from their position in a tree that realizes the exact working set property. This in turn allows us to approximate the total cost of the online *SeedTree* in comparison to the optimal offline algorithm with the same capacity. The approximation factor,  $2 - \log(f)$ , is intuitive: with less capacity in each level (lower values of levels' fractional occupancy), we need to put items further down.

**Lemma 4.** *SeedTree is MRU( $2 - \log(f)$ ) in expectation.*

*Proof.* For any given item  $v$  and time  $t$ , we show that  $E[\text{level}_t(v)] \leq \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor + 2 - \log(f)$  remains true, considering move-to-the-root and push-down phases. As can be seen in Line 8 of Algorithm 1, the item  $v$  might move down if the current level of  $v$  is lower than the level of the accessed item.

Let us denote the increase in the level from time  $t'$  to time  $t$  by a random variable  $D(t', t)$ . We express this increase in terms of an indicator random variable  $I(t', t, \ell)$  which denotes whether item  $v$  went down from level  $\ell$  during  $[t', t]$  or not. We know that:

$$D(t', t) = \sum_{\ell} I(t', t, \ell)$$

Let  $K$  denote the number of items accessed from a higher level, and let us write  $K = k_1 + \dots + k_{\lceil \frac{n}{c} \rceil}$ , where  $k_{\ell}$  means that  $k_{\ell}$  such accesses happened when item  $v$  was at level  $\ell$ . For the level  $\ell$ , based on the Observation 1 and Lemma 1 and the fact that each level contains  $f \cdot c \cdot 2^{\ell}$  items, we conclude  $v$  is being selected after  $k_{\ell} - 1$  accesses with probability  $(1 - \frac{1}{f \cdot c \cdot 2^{\ell}})^{k_{\ell}-1} \cdot (\frac{1}{f \cdot c \cdot 2^{\ell}})$ .

$$I(t', t, \ell) = \min(1, \sum_{k_{\ell}=0}^K (1 - \frac{1}{f \cdot c \cdot 2^{\ell}})^{k_{\ell}-1} \cdot (\frac{1}{f \cdot c \cdot 2^{\ell}}))$$

$$\begin{aligned}
&= \min(1, (\frac{1}{f \cdot c \cdot 2^\ell}) \cdot \sum_{k_\ell=0}^K (1 - \frac{1}{f \cdot c \cdot 2^\ell})^{k_\ell-1}) \\
&\leq \min(1, (\frac{K}{f \cdot c \cdot 2^\ell}))
\end{aligned}$$

Going back to our original goal of finding how many levels an item goes down during a time period  $[t', t]$ , we have:

$$\begin{aligned}
E[D(t', t)] &\leq \sum_{\ell} E[\min(1, (\frac{K}{f \cdot c \cdot 2^\ell}))] \\
&= \log(\frac{E[K]}{f \cdot c}) + 1 = \log(E[K]) - \log(c) - \log(f) + 1
\end{aligned}$$

The last equality comes from the fact that for  $\ell = \log(\frac{E[K]}{f \cdot c})$ , we have  $\frac{K}{f \cdot c \cdot 2^\ell} \leq 1$ , and for all larger values of  $\ell$ , the value will decrease exponentially with factors of two.

From Lemma 3 we know that the expected value of  $K$  is less than equal to  $2 \cdot \text{rank}_t(v)$ ; therefore, the expected increase is:

$$\begin{aligned}
E[D(t', t)] &\leq \log(2 \cdot \text{rank}_t(v)) - \log(c) - \log(f) + 1 \\
&= \log(\text{rank}_t(v)) - \log(c) + 2 - \log(f)
\end{aligned}$$

□

The following lemma shows the relation between the total cost of the online *SeedTree* and fractional occupancy  $f$ . The relation is natural: as  $f$  becomes smaller, the chance of finding a non-full node becomes larger, and thus fewer attempts are needed to find a non-full node.

**Lemma 5.** *The expected cost of SeedTree is less than equal to  $2 \cdot (\lceil \frac{1}{1-f} \rceil + 1)$  times the access cost.*

*Proof.* Let us consider the accessed item  $v$  at level  $\ell$ . In the first part of the algorithm, the move-to-the-root phase costs the same as the access, which is equal to traversing  $\ell$  edges. As the probability of a node being non-full is  $1 - f$  based on Observation 2, and as the choice of nodes is uniform based on Observation 1, only  $\lceil \frac{1}{1-f} \rceil$  iterations are needed during the push-down phase for finding a non-full node, each at cost  $2 \cdot \ell$ . Hence, given the linearity of expectation, we have:

$$\begin{aligned}
E[C_{ALG}] &= E[C_{ALG}^{\text{Access}} + C_{ALG}^{\text{Move-to-the-root}} + C_{ALG}^{\text{Push-down}}] \\
&\leq 2 \cdot (1 + \lceil \frac{1}{1-f} \rceil) \cdot \ell = 2 \cdot (1 + \lceil \frac{1}{1-f} \rceil) \cdot C_{ALG}^{\text{Access}}
\end{aligned}$$

□

We now describe why working set optimality is enough for dynamic optimality, given that reconfigurations do not cost much (which is proved in Lemma 5). Hence, any other form of optimality, such as key independent optimality or finger optimality is guaranteed automatically [11].

**Lemma 6.** *For any given  $c$ , an  $MRU(0)$  algorithm is  $(1+e)$  access competitive.*

*Proof.* The proof relies on the potential function argument. We describe a potential function at time  $t$  by  $\phi_t$ , and show that the change in the potential from time  $t$  to  $t+1$  is  $\Delta\phi_{t \rightarrow t+1}$ .

Our potential function at time  $t$ , counts the number of items that are misplaced in the tree of the optimal offline algorithm  $OPT$  with regard to their rank. (As the definition of  $MRU(0)$  indicates, there exists no inversion in such a tree, that is why we only focus on the number of inversions in  $OPT$ .) Concretely, we say a pair  $(v, u)$  is an inversion if  $\text{rank}_t(v) < \text{rank}_t(u)$  but  $\text{level}_t(v) > \text{level}_t(u)$ . We denote the number of items that have an inversion with item  $v$  at time  $t$  by  $\text{inv}_t(v)$ , and define  $B_t(v) = 1 + \frac{\text{inv}_t(v)}{c \cdot 2^{\text{level}_t(v)}}$ . Furthermore, define  $B_t = \prod_{v=1}^n B_t(v)$ . We define the potential function at time  $t$  as  $\phi_t = \log B_t$ . We assume that the online *SeedTree* rearranges its required items in the tree before the optimal algorithm's rearrangements. Let us first describe the change in potential due to rearrangement in the online *SeedTree* after accessing item  $\sigma_t = v$ . This change has the following effects:

- 1) Rank of the accessed item,  $v$ , has been set to 1.
- 2) Rank of other items in the tree might have been increased by at most 1.

Since the relative rank of items other than  $v$  does not change because of the second effect, it does not affect the number of inversions and hence the potential function. Therefore, we focus on the first effect. Since  $OPT$  has not changed its configuration, for all items  $u$  that are being stored in a lower level than  $v$  in the  $OPT$ , a single inversion is created, therefore we have  $B_{t+1}(u) = B_t(u) + \frac{1}{c \cdot 2^{\text{level}_t(u)}}$ . For the accessed item  $v$ , as its rank has changed to one, all of its inversions get deleted. The number of inversions for other items, except  $v$ , remains the same. Let us denote the number of items with lower level than  $v$  at time  $t$  by  $L_t(v)$  and partition the  $\prod_{i=1}^n B_{t+1}(i)$  into three parts as we discussed ( $v$ , items stored in a lower level than  $v$ , and other items denoted by set  $O_t(v)$ ):

$$\prod_{i=1}^n B_{t+1}(i) = B_{t+1}(v) \cdot \prod_{i \in L_t(v)} B_{t+1}(i) \cdot \prod_{i \in O_t(v)} B_{t+1}(i)$$

By rewriting  $B_{t+1}(i)$  in terms of  $B_t(i)$ , we get:

$$\prod_{i=1}^n B_{t+1}(i) = 1 \cdot \prod_{i \in L_t(v)} (B_t(i) + \frac{1}{c \cdot 2^{\text{level}_t(i)}}) \cdot \prod_{i \in O_t(v)} B_t(i)$$

Now let us look at potential due the first effect from time  $t$  to  $t+1$  by  $\Delta\phi_{t \rightarrow t+1}^1$ , and describe it in more detail:

$$\begin{aligned}
\Delta\phi_{t \rightarrow t+1}^1 &= \log B_{t+1} - \log B_t = \log \frac{B_{t+1}}{B_t} \\
&= \log \frac{\prod_{i=1}^n B_{t+1}(i)}{\prod_{i=1}^n B_t(i)} = \log \left( \frac{1}{B_t(v)} \cdot \frac{\prod_{i \in L_t(v)} (B_t(i) + \frac{1}{c \cdot 2^{\text{level}_t(i)}})}{\prod_{i \in L_t(v)} B_t(i)} \right) \\
&\leq \log \left( \frac{1}{B_t(v)} \cdot e^{|L_t(v)|} \right)
\end{aligned}$$



#### IV. APPLICATION IN RECONFIGURABLE DATACENTERS

*SeedTree* provides a fundamental self-adjusting structure which is useful in different settings. For example, it may be used to adapt the placement of containers in virtualized settings, in order to reduce communication costs. However, *SeedTree* can also be applied in reconfigurable networks in which links can be adapted. In the following, we describe how to use *SeedTree* in such a use case in more detail. In particular, we consider reconfigurable datacenters in which the connectivity between racks, or more specifically Top-of-the-Rack (ToR) switches, can be adjusted dynamically, e.g., based on optical circuit switches [6]. An optical switch provides a matching between racks, and accordingly, the model is known as a *matching model* in the literature [15]. In the following, we will show how a *SeedTree* with capacity  $c$  and fractional occupancy of  $f = \frac{1}{c}$  can be seen in terms of  $2 + c$  matchings, and how reconfigurations can be transformed to the matching model<sup>‡</sup>. We group these matchings into two sets:

- **Topological matchings:** consists of 2 *static* matchings, embedding the underlying binary tree of *SeedTree*. The first matching represents edges between a node and its left child (with the ID twice the ID of the node), and similarly the second matching for the right children (with the ID twice plus one of the ID of their parents). An example is depicted with solid edges in Figure 3.
- **Membership matchings:** has  $c$  *dynamic* matchings, connecting nodes to items inside them. If a node has more than one item, the corresponding order of items to matchings is arbitrary. An example is shown with dotted edges in Figure 3.

Having the matchings in place, let us briefly discuss how search and reconfiguration operations are implemented. A search for an item starts at the node with ID 001, the root node. We then check membership matchings of this node. If they map to the item, we have found the node which contains the item, and our search was successful. Otherwise, we follow the edge determined by the hash of the item, going to the new possible node hosting the item. We repeat the process of checking membership matchings and going along topological matchings until we find the item. The item will be found, as it is stored in one of the nodes in the path determined by its hash value. Each step of moving an item can be implemented in the matching mode with only one edge removal and one edge addition in membership matchings.

#### V. EXPERIMENTAL EVALUATION

We complement our analytical results by evaluating *SeedTree* on multiple datasets. Concretely, we are interested in answering the following questions:

- Q1 How does the access cost of our algorithm compare to the statically-optimal algorithm (optimized based on frequencies) and a demand-oblivious algorithm?

<sup>‡</sup>The matching model considers perfect matchings only, however, in practice imperfect matchings can be enforced by ignore rules in switches.

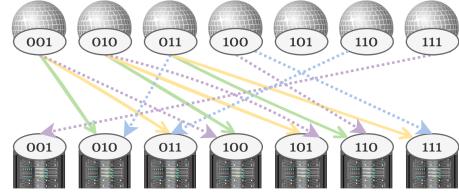


Fig. 3: A transformation from the example *SeedTree* shown in Figure 1, which has capacity  $c = 2$  and fractional occupancy of  $f = \frac{1}{2}$ . The disco balls on top represent the reconfigurable switches, and below are datacenter racks. Solid edges show structural matchings, and dotted edges represent membership matchings.

Q2 How does additional capacity improve the performance of the online *SeedTree*, given fixed fractional occupancy of each level?

Q3 What is the best initial fractional occupancy for the online *SeedTree*, given a fixed capacity?

Answers to these questions would help developers tune parameters of the *SeedTree* based on their requirements and needs. Before going through results, we describe the setup that we used: Our code is written in Python 3.6 and we used seaborn 0.11 [16] and Matplotlib 3.5 [17] libraries for visualization. Our programs were executed on a machine with 2x Intel Xeons E5-2697V3 SR1XF with 2.6 GHz, 14 cores each, and a total of 128 GB DDR4 RAM.

##### A. Input

- **Real-world dataset:** Our real-world dataset is communications between servers inside three different Facebook clusters, obtained from [1]. We post-processed this dataset for single-source communications. Among all possible sources, we chose the most frequent source.

- **Synthetic dataset:** We use the Markovian model discussed in [1], [18] for generating sequences based on a temporal locality parameter which ranges from 0 (uniform distribution, no locality) to 0.9 (high temporal locality). Our synthetic input consists of 65,535 items and 1 million requests. For generating such a dataset, we start from a random sample of items. We post-process this sequence, overwriting each request with the previous request with the probability determined by our temporal locality parameter. After that, we execute the second post-processing to ensure that exactly 65,535 items are in the final trace.

##### B. Algorithm setup

We use SHA-512 [19] from the *hashlib*-library as the hash function in our implementation, approximating the uniform distribution for generating addresses of items. In order to store items in a node we used a linked list, and when we move an item to a node that is already full with other items, items are stored in a temporary buffer. We assume starting from a pre-filled tree with items, a tree which respects the fractional occupancy parameter.

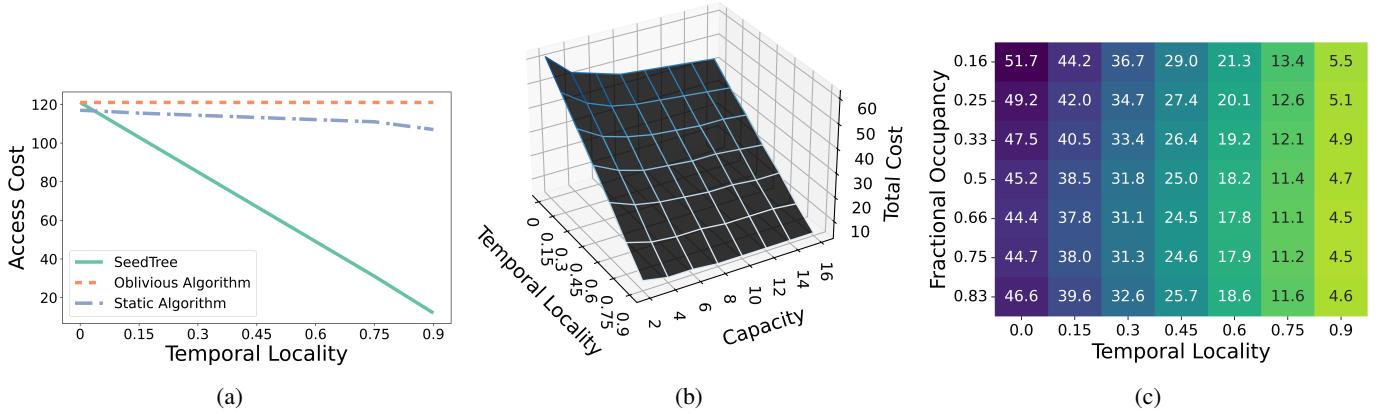


Fig. 4: Improvements in the performance of *SeedTree* by fine-tuning parameters. Figures are generated using the synthetic dataset with various locality values. (4a) Comparing the access cost of the *SeedTree* with fractional occupancy  $f = \frac{1}{2}$  to the best possible static algorithm and the demand-oblivious algorithm, all given capacity  $c = 4$ . Access costs are divided by 100 thousands. (4b) The effect of increasing capacity of nodes and temporal locality of input on the total cost of the algorithm. The fractional occupancy is set to  $f = \frac{1}{2}$  for all capacities. Total costs are divided by 1 million for this plot. (4c) Tradeoff between the total cost and the fractional occupancy, given a range of temporal localities. The capacity of nodes is set to 12. The number in each cell represents the cost, which are divided by 1 million.

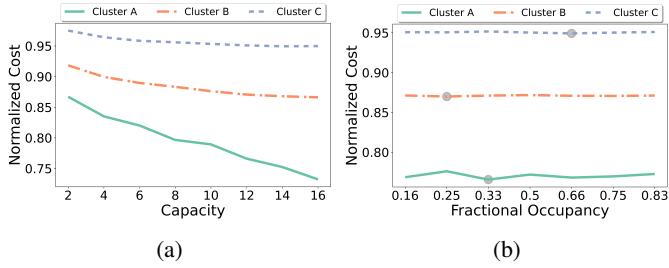


Fig. 5: Improvements in the normalized access cost of the algorithm by changing *SeedTree* parameters. These results are obtained based on communications of the most frequent source from three clusters of the real-world dataset. Costs are normalized by the cost of the demand-oblivious algorithm. (5a) Changes in the normalized cost by varying capacity. Fractional occupancy is set to  $f = \frac{1}{2}$ . (5b) Changes in the normalized cost by varying fractional occupancy. Gray dots show the minimum values. Capacity of nodes is set to 12.

In our experiments, we range the capacities ( $c$ ) from 2 to 16, and the fractional occupancies ( $f$ ) from 0.16 to 0.83. Due to the random nature of our algorithms and input generations, we repeat each experiment up to 100 times to ensure consistency in our results.

### C. Results

The performance of *SeedTree* improves significantly with the increased temporal locality, as can be seen in Figure 4. Furthermore, we have the following empirical answers to questions proposed at the beginning of this section:

A1: The *SeedTree* improves the access cost significantly, with increased temporal locality, as shown in Figures 4a,

which compares the access cost of *SeedTree* to static and demand-oblivious algorithms.

- A2: As the Figures 4b and 5a show, increasing capacity reduces the cost of the algorithm. However, as we can see, this increase slows down beyond capacity to 8, and hence this value can be considered as the best option for practical purposes.
- A3: As discussed at the end of the §III and can be seen in Figures 4c and 5b, the lowest cost can be achieved with fractions higher or lower than  $\frac{1}{2}$ , but  $f = \frac{1}{2}$  is near optimal in most scenarios.

### VI. ADDITIONAL RELATED WORK

Self-adjusting lists and trees have already been studied intensively in the context of data structures. The pioneering work is by Sleator and Tarjan [20], who initiated the study of the dynamic list update problems and who also introduced the move-to-front algorithm, inspiring many deterministic [21], [22] and randomized [23]–[26] approaches for datastructures, as well as other variations of the problem [27].

Self-adjusting binary search trees also aim to keep recently used elements close to the root, similarly to our approach in this paper (a summary of results is in Table I). However, adjustments in binary search trees are based on rotations rather than the movement of items between different nodes. One of the well-known self-adjusting binary search trees is the splay tree [9], although it is still unknown whether this tree is dynamically optimal; the problem is still open also for recent variations such as Zipper Tree [31], Multi Splay Tree [32] and Chain Splay [33] which improve the  $O(\log n)$  competitive ratio of the splay tree to  $O(\log \log n)$ . For Tango Trees [29], a matching  $\Omega(\log \log n)$  lower bound is known. We also know that if we allow for free rotations after access, dynamic

Data Structure	Operation	Ratio	Search
Splay Tree [9]	Rotation	$O(\log n)$	Yes
Greedy Future [28]	Rotation	$O(\log n)$	Yes
Tango Tree [29]	Rotation	$\theta(\log \log n)$	Yes
Adaptive Huffman [30]	Subtree swap	$\theta(1)$	No
Push-down Tree [10]	Item swap	$\theta(1)$	No
SeedTree	Item movement	$\theta(1)$	Yes

TABLE I: Comparison of properties of self-adjusting tree data structures. The best known competitive ratio (to this date) is in terms of the data structure’s respective cost model and optimal offline algorithm. We note that none of the above trees considers additional capacity, except for our model.

optimally becomes possible [34]. We also point out that some of these structures, in particular, multi splay tree and chain splay, benefitted from additional memory as well, however, there it is used differently, namely toward saving additional attributes for each node. Another variation which was first proposed by Lucas [28] in 1988 is called Greedy Future. This tree first received attention as an offline binary search tree algorithm [35], [36], but then an  $O(\log n)$  amortized time in online settings was suggested by Fox [37]. Greedy Future has motivated researchers to take a geometric view of online binary search trees [36], [38]. We note that in contrast to binary search trees, our local tree does not require an ordering of the items in the left and right subtrees of a node.

Self-adjusting trees have also been explored in the context of coding, where for example adaptive Huffman coding [30], [39]–[42] is used to minimize the depth of most frequent items. The reconfiguration cost, however, is different: in adaptive Huffman algorithms, two subtrees might be swapped at the cost of one.

A few data structures have tried to achieve a better competitive ratio by expanding and altering binary search trees (see Table II for a summary): The first example, PokeTree [43], adds extra pointers between the internal nodes of the tree and achieves an  $O(\log \log n)$  competitive ratio in comparison to an optimal binary search tree. There are also self-adjusting data structures based on skip lists [44], [45], which have been introduced as an alternative for balanced trees that enforce probabilistic balancing instead. A biased version of skip lists was considered in [46], and later on, a statically optimal variation was given in [47] and a dynamic optimal version in a restricted model in [48]. Another example is Iacono’s working set structure [49] which combines a series of self-adjusting balanced binary search trees and deques, achieving a worst-case running time of  $O(\log n)$ , however, it lacks the dynamic optimality property. We are not aware of any work exploring augmentations to improve the competitive ratio of these data structures.

Our work is also motivated by emerging self-adjusting datacenter networks. Recent optical communication technologies enable datacenters to be reconfigured quickly and frequently [8], [18], [50]–[58], see [59] for a recent survey. The datacenter application mentioned in our paper is based on the matching model proposed by [15]. Recently [60] introduced

Data Structure	Structure	Ratio
Iacono’s structure [49]	Trees & deques	$O(\log n)$
Skip List [44]	Linked lists	$O(\log n)$
PokeTree [43]	Tree & dynamic links	$O(\log \log n)$
SeedTree	Tree	$\theta(1)$

TABLE II: Comparison with other self-adjusting data structures that support local-search. The best known competitive ratio (to this date) is in terms of the data structure’s respective cost model and optimal offline algorithm. We note that none of the other data structures considers capacity in their design.

an online algorithm for constructing self-adjusting networks based on this model, however the authors do not provide dynamic optimality proof for their method.

It has been shown that demand-aware and self-adjusting datacenter networks can be built from individual trees [61], called ego-trees, which are used in many network designs [8], [50], [62], [63], and also motivate our model. However, until now it was an open problem how to design self-adjusting and constant-competitive trees that support local routing and adjustments, a desirable property in dynamic settings.

Last but not least, our work also features interesting connections to peer-to-peer networks [12], [64]. It is known that consistent hashing with previously assigned and fixed capacities allows for significantly improved load balancing [13], [14], which has interesting applications and is used, e.g., in Vimeo’s streaming service [65] and in Google’s cloud service [13]. Although these approaches benefit from data structures with capacity, these approaches are not demand-aware.

## VII. CONCLUSION AND FUTURE WORK

This paper presented and evaluated a self-adjusting and local tree, *SeedTree*, which adapts towards the workload in an online, constant-competitive manner. *SeedTree* supports a capacity augmentation approach, while providing local routing, which can be useful for other self-adjusting structures and applications as well. We showed a transformation of our algorithm into the matching model for application in reconfigurable datacenters, and evaluated our algorithm on synthetic and real-world communication traces. The code used for our experimental evaluation is available at [github.com/inet-tub/SeedTree](https://github.com/inet-tub/SeedTree).

We believe that our work opens several interesting avenues for future research. In particular, while we so far focused on randomized approaches, it would be interesting to explore deterministic variants of *SeedTree*. Furthermore, while trees are a fundamental building block toward more complex networks (as they, e.g., arise in datacenters today), it remains to design and evaluate networks based on *SeedTree*.

## REFERENCES

- [1] C. Avin, M. Ghobadi, C. Griner, and S. Schmid, “On the complexity of traffic traces and implications,” in *ACM SIGMETRICS*, 2020.
- [2] T. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding data center traffic characteristics,” *ACM SIGCOMM CCR*, 2010.
- [3] O. Michel, R. Bifulco, G. Retvari, and S. Schmid, “The programmable data plane: Abstractions, architectures, algorithms, and applications,” in *ACM CSUR*, 2021.

